VISIT…

Thinking in Tkinter

by Stephen Ferg (steve@ ferg.org)
revised: 2005-07-17

This file contains the source code for all of the files in the *Thinking in Tkinter* series.

If you print this file with a small font, you should be able to print it in portrait mode without truncating the ends of any of the lines of code. If you print it, and find that you ARE truncating lines of code, consider printing it in a smaller font, or printing it in landscape mode.

Printed size is approximately 40 to 60 pages, depending on your print settings. You can see what the printed result will look like, and how many pages it will require, by looking at the results of *Print Preview* before you actually print the file.

tt000.py

```
Subject: "Thinking in Tkinter"
Author : Stephen Ferg (steve@ferg.org)
```

About " Thinking In Tkinter "

I've been trying to teach myself Tkinter out of various books, and I'm finding it more difficult than I think it should be.

The problem is that the authors of the books want to rush into telling me about all of the widgets in the Tkinter toolbox, but never really pause to explain basic concepts. They don't explain how to "think in Tkinter".

Here are a few short programs that begin to explain how to think in Tkinter. In them, I don't attempt to catalog all of the types of widgets, attributes, and methods that are available in Tkinter. and I certainly don't try to provide a comprehensive introduction to Tkinter. I just try to get you started down the road of understanding some basic Tkinter concepts.

Note that the discussion is devoted exclusively to the Tkinter pack (or "packer") geometry manager. There is no discussion of the grid or place geometry managers.

The Four Basic Gui-programming Tasks

When you develop a user interface (UI) there is a standard set of tasks that you must accomplish.

1) You must specify how you want the UI to *look*. That is, you must write code that determines what the user will see on the computer screen.

2) You must decide what you want the UI to *do*. That is, you must write routines that accomplish the tasks of the program.

3) You must associate the "looking" with the "doing". That is, you must write code that associates the things that the user sees on the screen with the routines that you have written to perform the program's tasks.

4) finally, you must write code that sits and waits for input from the user.

Some Gui-programming Jargon

GUI (graphical user interface) programming has some special jargon associated with these basic tasks.

1) We specify how we want a GUI to look by describing the "widgets" that we want it to display, and their spatial relationships (i.e. whether one widget is above or below, or to the right or left, of other widgets). The word "widget" is a nonsense word that has become the common term for "graphical user interface component". Widgets include things such as windows, buttons, menus and menu items, icons, drop-down lists, scroll bars, and so on.

2) The routines that actually do the work of the GUI are called "callback handlers" or "event handlers". "Events" are input events such as mouse clicks or presses of a key on the keyboard. These routines are called "handlers" because they "handle" (that is, respond to) such events.

3) Associating an event handler with a widget is called "binding". Roughly, the process of binding involves associating three different things:

```
(a) a type of event (e.g. a click of the left mouse button,
    or a press of the ENTER key on the keyboard),
(b) a widget (e.g. a button), and
(c) an event-handler routine.
```

For example, we might bind (a) a single-click of the left mouse button on (b) the "CLOSE" button/widget on the screen to (c) the "closeProgram" routine, which closes the window and shuts down the program.

4) The code that sits and waits for input is called the "event loop".

About The Event Loop

If you believe the movies, every small town has a little old lady who spends all of her time at her front window, just WATCHING. She sees everything that goes on in the neighborhood. A lot of what she sees is uninteresting of course -- just people going to and fro in the street. But some of it is interesting -- like a big fight between the newly-wed couple in the house across the street. When interesting events happen, the watchdog lady immediately is on the phone with the news to the police or to her neighbors.

The event loop is a lot like this watchdog lady. The event loop spends all of its time watching events go by, and it sees all of them. Most of the events are uninteresting, and when it sees them, it does nothing. But when it sees something interesting -- an event that it knows is interesting, because an event handler has been bound to the event -- then it immediately calls up the event handler and lets it know that the event has happened.

Program Behavior

This program eases you into user-interace programming by showing how these basic concepts are

implemented in a very simple program. This program doesn't use Tkinter or any form of GUI programming. It just puts up a menu on the console, and gets simple keyboard input. Even so, as you can see, it does the four basic tasks of user-interface programming.

[revised: 2003-02-23]

Program Source Code

```
#----- task 2:  define the event handler routines ---------------------
def handle_A():
    print "Wrong! Try again!"

def handle_B():
    print "Absolutely right!  Trillium is a kind of flower!"

def handle_C():
    print "Wrong! Try again!"

# ------------ task 1: define the appearance of the screen ------------
print "\n"*100   # clear the screen
print "            VERY CHALLENGING GUESSING GAME"
print "============================================================"
print "Press the letter of your answer, then the ENTER key."
print
print "    A.  Animal"
print "    B.  Vegetable"
print "    C.  Mineral"
print
print "    X.  Exit from this program"
print
print "============================================================"
print "What kind of thing is 'Trillium'?"
print

# ---- task 4: the event loop.  We loop forever, observing events. ---
while 1:

    # We observe the next event
    answer = raw_input().upper()

    # -------------------------------------------------------
    # Task 3: Associate interesting keyboard events with their
    # event handlers.  A simple form of binding.
    # -------------------------------------------------------
    if answer == "A": handle_A()
    if answer == "B": handle_B()
    if answer == "C": handle_C()
    if answer == "X":
        # clear the screen and exit the event loop
        print "\n"*100
        break

    # Note that any other events are uninteresting, and are ignored
```

tt010.py

The Simplest Possible Tkinter Program -- Three Statements!

Of the four basic GUI tasks that we discussed in the last program, this program does only one -- it runs the event loop.

(1) The first statement imports Tkinter, so that it is available for use. Note that the form of the import ("from Tkinter import *") means that we will not have to qualify anything that we get from Tkinter with a "Tkinter." prefix.

(2) The second statement creates a "toplevel" window. Technically, what the the second statement is doing, is creating an instance of the class "Tkinter.Tk".

This toplevel window is the highest-level GUI component in any Tkinter application. By convention, the toplevel window is usually named "root".

(3) The third statement executes the "mainloop" (that is, the event loop) method of the "root" object. As the mainloop runs, it waits for events to happen in root. If an event occurs, then it is handled and the loop continues running, waiting for the next evernt. The loop continues to execute until a "destroy" event happens to the root window. A "destroy" event is one that closes a window. When the root is destroyed, the window is closed and the event loop is exited.

Program Behavior

When you run this program, you will see that (thanks to Tk) the toplevel window automatically comes furnished with widgets to minimize, maximize, and close the window. Try them -- you'll see that they really do work.

Clicking on the "close" widget (the "x" in a box, at the right of the title bar) generates a "destroy" event. The destroy event terminates the main event loop. And since there are no statements after "root.mainloop()", the program has nothing more to do, and ends.

[revised: 2003-02-23]

Program Source Code

```
from Tkinter import * ### (1)

root = Tk()           ### (2)
root.mainloop()       ### (3)
```

tt020.py

Now we tackle another of the four main GUI tasks -- specifying how the GUI should look.

In this program, we introduce three major concepts of Tkinter programming:

```
* creating a GUI object and associating it with its parent
* packing
* containers vs. widgets
```

From now on, I'm going to distinguish between a container component and a widget. As I will be using the terms, a "widget" is a GUI component that (usually) is visible and does things. A "container" in contrast is simply a container -- a basket, as it were -- into which we can put widgets.

Tkinter provides a number of containers. "Canvas" is a container for drawing applications. The most frequently used container is a "frame".

Frames are provided by Tkinter in a class called "Frame". An expression like:

```
Frame(myParent)
```

creates an instance of the Frame class (that is, it creates a frame), and associates the frame instance with its parent, myParent. Or another way of looking at it is: such an expression adds a child frame to the myParent component.

So in this program, statement (1):

```
myContainer1 = Frame(myParent)
```

creates a frame whose parent is myParent (that is, root), and gives it the name "myContainer1". In short, it creates a container into which we can put widgets. (We won't put any widgets into in this program. We'll do that in later programs.)

Note that the parent/child relationship here is a LOGICAL one, not a visual one. This relationship exists to support such things as the destroy event -- so that when a parent component (such as the root) is destroyed, the parent knows who its children are, and can destroy them before destroying itself.

(2) The next statement "packs" myContainer1.

```
myContainer1.pack()
```

Simply put, "packing" is a process of setting up a VISUAL relationship between a GUI component and its parent. If you don't pack a component, you will never see it.

"Pack" invokes the Tkinter "pack" geometry manager. A geometry manager is essentially an API -- a way of talking to Tkinter -- for telling Tkinter how you want containers and widgets to be visually presented. Tkinter supports three geometry managers: pack, grid, and place. Pack (and to a lesser extent) grid are the most widely used, because they are the easiest to use. All of the examples in

"Thinking in Tkinter" use the pack geometry manager.

So here you see a basic pattern for Tkinter programming that we will see over and over again.

```
(1) an instance (of a widget or a container) is created, and associated with its parent
(2) the instance is packed.
```

Program Behavior

When you run this program, it will look very much like the previous one, except that there will be less to see. That is because ...

Frames Are Elastic

A frame is basically a container. The interior of a container -- the "space" as it were, inside the container -- is called the "cavity". ("Cavity" is a technical term that Tkinter gets from Tk.)

This cavity is "stretchy" or elastic, like a rubber band. Unless you specify an minimum or maximum size for the frame, the cavity will stretch or shrink to accommodate whatever is placed inside the frame.

In the previous program, because we hadn't put anything into it, the root displayed itself with its default size.

But in this program, we *have* put something into the root's cavity -- we have put Container1 into it. So the root frame shrinks to accommodate the size of Container1. But since we haven't put any widgets into Container1, and we haven't specified a minimum size for Container1, the root's cavity shrinks down to nothing. That is why there is nothing to see below the title bar.

In the next few programs, we will put widgets and other containers into Container1, and you will see how Container1 stretches to accommodate them.

[revised: 2003-02-24]

Program Source Code

```
from Tkinter import *

root = Tk()

myContainer1 = Frame(root)  ### (1)
myContainer1.pack()         ### (2)

root.mainloop()
```

tt030.py

In this program, we create our first widget, and put it into myContainer1.

(1) The widget will be a button -- that is, it will be an instance of the Tkinter "Button" class. The statement:

```
button1 = Button(myContainer1)
```

creates the button, gives it the name "button1", and associates it with its parent, the container object called myContainer1.

(2)(3) Widgets have many attributes, which are stored in their local namespace dictionary. Button widgets have attributes to control their size, their foreground and background colors, the text that they display, how their borders look, and so on. In this example, we will set just two of button1's attributes: the background color and the text. We do it by setting the values in the button's dictionary with the keys "text" and "background".

```
button1["text"]= "Hello, World!"
button1["background"] = "green"
```

(4) And of course, we pack button1.

```
button1.pack()
```

Some Useful Technical Terminology

Sometimes the relationship between a container and the widget(s) that it contains is referred to as a "parent/child" relationship. It is also referred to as a "master/slave" relationship.

Program Behavior

When you run this program, you should see that Container1 now contains a green button with the text "Hello, World!". When you click on it, it won't do anything, because we haven't yet specified what we want to happen when the button is clicked. (We'll do that later.)

For now, you will have to close the window, as before, by clicking the CLOSE icon on the title bar.

Note how myContainer1 has stretched to accommodate button1.

[revised: 2002-10-01]

Program Source Code

```
from Tkinter import *

root = Tk()
```

```
myContainer1 = Frame(root)
myContainer1.pack()

button1 = Button(myContainer1)       ### (1)
button1["text"]= "Hello, World!"     ### (2)
button1["background"] = "green"      ### (3)
button1.pack()                       ### (4)

root.mainloop()
```

tt035.py

Using A Class Structure

In this program, we introduce the concept of structuring a Tkinter application as a set of classes.

In this program, we have added a class called MyApp and moved some of the code from the previous program into its constructor (__init__) method. In this restructured version of the program, we do 3 different things:

(1) In our code, we define a class (MyApp) that defines how we want our GUI to look. It defines the way we want our GUI to look and the kinds of things that we want to do with it. All of this code is put into the constructor (__init__) method of the class. (1a)

(2) When the program executes, the first thing it does is to create an instance of the class. The statement that creates the instance is

```
myapp = MyApp(root)
```

Note that the name of the class is "MyApp" (note the capitalization) and the name of the instance is "myapp" (note the lack of capitalization).

Note also that this statement passes "root" as an argument into the constructor method (__init__) of MyApp. The constructor method recognizes the root under the name "myParent". (1a)

(3) Finally, we run mainloop on the root.

Why Structure Your Application As A Class?

One of the reasons to use a class structure in your program is simply to control the program better. A program that is structured into classes is probably -- especially if it is a very big program -- a lot easier to understand than one that is unstructured.

A more important consideration is that structuring your application as a class helps you to avoid the use of global variables. Eventually, as your program grows, you will probably want some of your event handlers to be able to share information among themselves. One way is to use global variables, but that is a very messy techniqe. A much better way is to use instance (that is, "self." variables), and for that you must give your application a class structure. We will explore this issue in a later program in this series.

When To Introduce Class Structuring

We've introduced the notion of a class structure for Tkinter programs early, in order to explain it and then move on to other matters. But in actual development, you may choose to proceed differently.

In many cases, a Tkinter program starts as a simple script. All of the code is inline, as in our previous program. Then, as you understand new dimensions of the application, the programs grows. After a

while, you have a LOT of code. You may have started to use global variables... maybe a LOT of global variables. The program starts to become difficult to understand and modify. When that happens, it is time to refactor your program, and to re-structure it using classes.

On the other hand, if you are comfortable with classes, and have a pretty good idea of the final shape of your program, you may choose to structure your program using classes from the very beginning.

But on the other hand (back to the first hand?), early in the development process (Gerrit Muller has observed) often you don't yet know the best class structure to use -- early in the process, you simply don't have a clear enough understanding of the problem and the solution. Starting to use classes too early in the process can introduce a lot of unnecessary structure that merely clutters up the code, hinders understanding, and eventually requires more refactoring.

So it is pretty much a matter of individual taste and experience and prevailing circumstances. Do what feels right for you. And -- no matter what you choose to do -- don't be afraid to do some serious refactoring when you need to.

Program Behavior

When you run this program, it will look exactly like the previous one. No functionality has been changed -- only how the code is structured.

[revised: 2003-02-23]

Program Source Code

```
from Tkinter import *

class MyApp:                            ### (1)
    def __init__(self, myParent):       ### (1a)
        self.myContainer1 = Frame(myParent)
        self.myContainer1.pack()

        self.button1 = Button(self.myContainer1)
        self.button1["text"]= "Hello, World!"
        self.button1["background"] = "green"
        self.button1.pack()

root = Tk()
myapp = MyApp(root)  ### (2)
root.mainloop()      ### (3)
```

tt040.py

(1) In the previous program, we created a button object, button1, and then set its text and backround color in a fairly straightforward way.

```
self.button1["text"]= "Hello, World!"
self.button1["background"] = "green"
```

In this program, we add three more button to Container1, using slightly different methods.

(2) For button2, the process is essentially the same as for button1, but instead of accessing the button's dictionary, we use the button's built-in "configure" method.

(3) For button3, we see that the configure method can take multiple keyword arguments, so we can set multiple options in a single statement.

(4) In the previous examples, setting up the button has been a two-step process: first we create the button, then we set its properties. But it is possible to specify the button's properties at the time that we create it. The "Button" widget (like all widgets) expects its first argument to be its parent. This is a positional argument, not a keyword argument. But after that, you can, if you wish, add one or more keyword arguments that specify properties of the widget.

Program Behavior

When you run this program, you should see that Container1 now contains, in addition to the original green button, three more buttons.

Note how myContainer1 has stretched to accommodate these other buttons.

Note also that the buttons are stacked on top of each other. In the next program, we will see why they arrange themselves this way, and see how to arrange them differently.

Program Source Code

```
from Tkinter import *

class MyApp:
    def __init__(self, parent):
        self.myContainer1 = Frame(parent)
        self.myContainer1.pack()

        self.button1 = Button(self.myContainer1)
        self.button1["text"] = "Hello, World!"   ### (1)
        self.button1["background"] = "green"     ### (1)
        self.button1.pack()

        self.button2 = Button(self.myContainer1)
        self.button2.configure(text="Off to join the circus!") ### (2)
        self.button2.configure(background="tan")               ### (2)
        self.button2.pack()
```

```
        self.button3 = Button(self.myContainer1)
        self.button3.configure(text="Join me?", background="cyan")  ### (3)
        self.button3.pack()

        self.button4 = Button(self.myContainer1, text="Goodbye!", background="red") ### (4)
        self.button4.pack()


root = Tk()
myapp = MyApp(root)
root.mainloop()
```

tt050.py

In the last program, we saw two buttons, stacked on top of each other. Probably, however, we'd like to see them side-by-side. In this program, we do that, and we start to see what we can do with pack().

(1) (2) (3) (4)

Packing is a way of controlling the VISUAL relationship of components. So what we are going to do now is to use the pack "side" option to put the buttons side-by-side rather than stacked on top of each other. We do it with the "side" argument to the pack() statement, for example:

```
self.button1.pack(side=LEFT)
```

Note that LEFT (like RIGHT, TOP, and BOTTOM) are user-friendly constants defined in Tkinter. That is, "LEFT" is actually "Tkinter.LEFT" -- but because of the way that we imported Tkinter, we don't need to supply the "Tkinter." prefix.

Why The Buttons Were Stacked Vertically In The Last Program

As you remember, in the last program, we just packed the buttons without specifying any "side" option, and the buttons packed on top of each other. That is because the default "side" option is "side=TOP".

So when we packed button1, it was packed at the top of the cavity inside of myContainer1. That left the cavity for myContainer1 positioned below button1. Then we packed button2. It was packed at the the top of the cavity, which means that it was positioned immediately below button1. And the cavity is now positioned below button2.

If we had packed the buttons in a different order -- for example, if we had packed button2 first, and then packed button1 -- their positions would have been reversed, and button2 would have been on top.

So, as you can see, one of the ways that you can control the appearance of your GUI is by controlling the order in which you pack widgets inside containers.

Some Technical Terminology -- "orientation"

"Vertical" orientation includes the TOP and BOTTOM sides. "Horizonal" orientation includes the LEFT and RIGHT sides.

When you are packing widgets and containers, it is possible to mix the two orientations. For example, we could have packed one button with a vertical orientation (say, TOP) and the other button with a horizontal orientation (say, LEFT).

But mixing orientations inside a container this way is not a good idea. If you use mixed orientations, then predicting what the final result will look like is difficult, and you will likely be surprised by the way the GUI looks if the window is re-sized.

So it is a good design practice never to mix orientations with the same container. The way to handle

complicated GUIs, where you really do want to use multiple orientations, is to nest containers within containers. We'll explore that topic in a later program.

Program Behavior

When you run this program, you will now see the two buttons, side by side.

[revised: 2002-10-01]

Program Source Code

```
from Tkinter import *

class MyApp:
    def __init__(self, parent):
        self.myContainer1 = Frame(parent)
        self.myContainer1.pack()

        self.button1 = Button(self.myContainer1)
        self.button1["text"]= "Hello, World!"
        self.button1["background"] = "green"
        self.button1.pack(side=LEFT)    ### (1)


        self.button2 = Button(self.myContainer1)
        self.button2.configure(text="Off to join the circus!")
        self.button2.configure(background="tan")
        self.button2.pack(side=LEFT)     ### (2)


        self.button3 = Button(self.myContainer1)
        self.button3.configure(text="Join me?", background="cyan")
        self.button3.pack(side=LEFT)      ### (3)

        self.button4 = Button(self.myContainer1, text="Goodbye!", background="red")
        self.button4.pack(side=LEFT)      ### (4)


root = Tk()
myapp = MyApp(root)
root.mainloop()
```

tt060.py

Now it is time to get our buttons to do something. We turn our attention to the last two (or our original four) basic GUI tasks -- writing event handler routines to do the actual work of our program, and binding the event handler routines to widgets and events.

Note that in this program we have abandoned all the buttons that we created in our last program, and have returned to a simpler situation in which our GUI contains only two buttons: "OK" and "Cancel".

As you will remember from the discussion in our first program, one of the basic GUI task is "binding". "Binding" is the process of defining a connection or relationship between (usually):

```
* a widget,
* a type of event, and
* an "event handler".
```

An "event handler" is a method or subroutine that handles events when they occur. [In Java, event handlers are called "listeners". I like this name, because it suggests exactly what they do -- "listen" for events, and respond to them.]

In Tkinter, the way that you create this binding is through the bind() method that is a feature of all Tkinter widgets. You use the bind() method in a statement of the form:

```
widget.bind(event_type_name, event_handler_name)
```

This kind of binding is called "event binding".

[There is another way of binding an event_handler to a widget. It is called "command binding" and we will look at it a couple of programs from now. But for now, let's look at event binding. Once we understand what event binding is, it will make it easier to explain command binding.]

Before we begin, we need to point out a possible point of confusion. The word "button" can be used to refer to two entirely different things: (1) a button widget -- a GUI component that is displayed on the computer monitor -- and (2) a button on your mouse -- the kind of button that you press with your finger. In order to avoid confusion, I will usually try to distinguish them by referring to "button widget" or "mouse button" rather than simply to "button".

(1) We bind "<Button-1>" events (clicks of the left mouse button) on the button1 widget to the "self.button1Click" method. When button1 is left-clicked with the mouse, the self.button1Click() method will be invoked to handle the event.

(3) Note that, although they aren't specified on the "bind" operation, self.button1Click() will be passed two arguments. The first, of course, will be "self", which is always the first argument to all class methods in Python. The second will be an event object. This technique of binding and event (that is, using the bind() method) always implicitly passes an event object as an argument.

In Python/Tkinter, when an event occurs, it takes the form of an event object. An event object is

extremely useful, because it carries with it all sorts of useful information and methods. You can examine the event object to find out what kind of event occurred, the widget where it occurred, and other useful bits of information.

(4) So, what do we want to happen when button1 is clicked? Well, in this case we have it do something quite simple. It simply changes its color from green to yellow, and back again.

(2) Let's make button2 (the "Goodbye!" button) actually do some useful work. We will make it shut down the window. So we bind a left-mouse click in button2 to the button2Click() method, and

(6) We have the button2Click() method destroy the root window, the parent window of myapp. This will have a ripple effect, and will destroy all the children and descendents of the root. In short, every part of the GUI will be destroyed.

Of course, to do this, myapp has to know who its parent is. So (7) we add code to its constructor to allow myapp to remember its parent.

Program Behavior

When you run this program, you will see two buttons. Clicking on the "OK" button will change its color. Clicking on the "Cancel" button will shut down the application.

When our GUI is open, if you hit the TAB key on the keyboard, you will notice that the keyboard focus tabs between the two buttons. But if you hit the ENTER/RETURN key on the keyboard, nothing happens. That is because we have bound only mouse clicks, not keyboard events, to our buttons. Our next task will be to bind keyboard events to the buttons, also.

Finally, notice that because the text of one button is shorter than the text of the other, the two buttons are of different sizes. This is rather ugly. We will fix that in a later program.

[revised: 2002-10-01]

Program Source Code

```
from Tkinter import *

class MyApp:
    def __init__(self, parent):
        self.myParent = parent  ### (7) remember my parent, the root
        self.myContainer1 = Frame(parent)
        self.myContainer1.pack()

        self.button1 = Button(self.myContainer1)
        self.button1.configure(text="OK", background= "green")
        self.button1.pack(side=LEFT)
        self.button1.bind("<Button-1>", self.button1Click) ### (1)

        self.button2 = Button(self.myContainer1)
        self.button2.configure(text="Cancel", background="red")
        self.button2.pack(side=RIGHT)
        self.button2.bind("<Button-1>", self.button2Click) ### (2)

    def button1Click(self, event):    ### (3)
        if self.button1["background"] == "green": ### (4)
```

```
            self.button1["background"] = "yellow"
        else:
            self.button1["background"] = "green"

    def button2Click(self, event):  ### (5)
        self.myParent.destroy()     ### (6)


root = Tk()
myapp = MyApp(root)
root.mainloop()
```

tt070.py

In the previous program, you could make the buttons do something by clicking on them with the mouse, but you couldn't make them do something by pressing a key on the keyboard. In this program, we see how to make them react to keyboard events as well as mouse events.

First, we need the concept of "input focus", or simply "focus".

If you're familiar with Greek mythology (or if you saw the Disney animated movie "Hercules") you may remember the Fates. The Fates were three old women who controlled the destinies of men. Each human life was a thread in the hands of the Fates, and when they cut the thread, the life ended.

The remarkable thing about the Fates was that they shared only one eye among all three of them. The one with the eye had to do all of the seeing, and tell the other two what she saw. The eye could be passed from one Fate to another, so they could take turns seeing. And of course, if you could steal the eye, you had a MAJOR bargaining chip when negotiating with the Fates.

"Focus" is what allows the widgets on your GUI to see keyboard events. It is to the widgets on your GUI, what the single eye was to the Fates.

Only one widget at a time can have the focus, and the widget that "has focus" is the widget that sees, and responds to, keyboard events. "Setting focus" on a widget is the process of giving the focus to the widget.

In this program, for example, our GUI has two buttons: "OK" and "Cancel". Suppose I hit the RETURN button on the keyboard. Will that keypress "Return" event be seen by (or sent to) the "OK" button, indicating the user has accepted his choice? Or will the "Return" event be seen by (or sent to) the "Cancel" button, indicating that the user has cancelled the operation? It depends on where the "focus" is. That is, it depends on which (if any) of the buttons "has focus".

Like the Fates' eye, which could be passed from one Fate to another, focus can be passed from one GUI widget to another. There are several ways of passing, or moving, the focus from one widget to another. One way is with the mouse. You can "set focus" on a widget by clicking on the widget with the left mouse button. (At least, this model, which is called the "click to type" model, is the way it works on Windows and Macintosh, and in Tk and Tkinter. There are some systems that use a "focus follows mouse" convention in which the widget that is under the mouse automatically has focus, and no click is necessary. You can get the same effect in Tk by using the tk_focusFollowsMouse procedure.)

Another way to set focus is with the keyboard. The set of widgets that are capable of receiving the focus are stored in a circular list (the "traversal order") in the order in which the widgets were created. Hitting the TAB key on the keyboard moves the focus from its current location (which may be nowhere) to the next widget in the list. At the end of the list, the focus moves to the widget at the head of the list. And hitting SHIFT+TAB moves the focus backward, rather than forward, in the list.

When a GUI button has focus, the fact that it has focus is shown by a small dotted box around the text of the button. Here's how to see it. Run our previous program. When the program starts, and the GUI

displays, neither of the buttons has focus, so you don't see the dotted box. Now hit the TAB key. You will see the little dotted box appear around the left button, showing that focus has been given to it. Now hit the TAB key again, and again. You will see how the focus jumps to the next button, and when it reaches the last button, it wraps around again to the first one. (Since the program shows only two buttons, the effect is that the focus jumps back and forth between the two buttons.)

(0) In this program, we would like the OK button to have focus from the very beginning. So we use the "focus_force()" method, which forces the focus to go to the OK button. When you run this program, you will see that the OK button has focus from the time the application starts.

In the last program, our buttons responded to only one keyboard event -- a keyprees of the TAB key -- which moved the focus back and forth between the two buttons. But if you hit the ENTER/RETURN key on the keyboard, nothing happened. That is because we had bound only mouse clicks, not keyboard events, to our buttons.

In this program we will also bind keyboard events to the buttons.

(1) (2) The statements to bind keyboard events to the buttons are quite simple -- they have the same format as statements to bind mouse events. The only difference is that the name of the event is the name of a keyboard event (in this case, "<Return>") rather than a mouse event.

We want a press of the RETURN key on the keyboard and a click of the left mouse button to have the same effect on the widget, so we bind the same event handler to both types of events.

This program shows that you can bind multiple types of events to a single widget (such as a button). And you can bind multiple <widget, event> combinations to the same event handler.

(3) (4) Now that our button widgets respond to multiple kinds of events, we can demonstrate how to retrieve information from an event object. What we will do is to pass the event objects to (5) a "report_event" routine that will (6) print out information about the event that it obtains from the event's attributes.

Note that in order to see this information printed out on the console, you must run this program using python (not pythonw) from a console window.

Program Behavior

When you run this program, you will see two buttons. Clicking on the left button, or pressing the RETURN key when the button has the keyboard focus, will change its color. Clicking on the right button, or pressing the RETURN key when the button has the keyboard focus, will shut down the application. For any of these keyboard or mouse events, you should see a printed message giving the time of the event and describing the event.

[Revised: 2002-09-26]

Program Source Code

```
from Tkinter import *

class MyApp:
    def __init__(self, parent):
        self.myParent = parent
```

```
        self.myContainer1 = Frame(parent)
        self.myContainer1.pack()

        self.button1 = Button(self.myContainer1)
        self.button1.configure(text="OK", background= "green")
        self.button1.pack(side=LEFT)
        self.button1.focus_force()          ### (0)
        self.button1.bind("<Button-1>", self.button1Click)
        self.button1.bind("<Return>", self.button1Click)  ### (1)

        self.button2 = Button(self.myContainer1)
        self.button2.configure(text="Cancel", background="red")
        self.button2.pack(side=RIGHT)
        self.button2.bind("<Button-1>", self.button2Click)
        self.button2.bind("<Return>", self.button2Click)  ### (2)

    def button1Click(self, event):
        report_event(event)         ### (3)
        if self.button1["background"] == "green":
            self.button1["background"] = "yellow"
        else:
            self.button1["background"] = "green"

    def button2Click(self, event):
        report_event(event)   ### (4)
        self.myParent.destroy()

def report_event(event):     ### (5)
    """Print a description of an event, based on its attributes.
    """
    event_name = {"2": "KeyPress", "4": "ButtonPress"}
    print "Time:", str(event.time)   ### (6)
    print "EventType=" + str(event.type), \
        event_name[str(event.type)],\
        "EventWidgetId=" + str(event.widget), \
        "EventKeySymbol=" + str(event.keysym)


root = Tk()
myapp = MyApp(root)
root.mainloop()
```

tt074.py

In an earlier program, we introduced "event binding". There is another way of binding an event_handler to a widget. It is called "command binding" and we will look at it in this program.

Command Binding

You will remember that in our previous programs, we bound the "<Button-1>" mouse event to our button widget. "Button" is another name for a "ButtonPress" mouse event. And a "ButtonPress" mouse event is different from a "ButtonRelease" mouse event. The "ButtonPress" event is the act of pushing down on the mouse button, BUT NOT RELEASING IT. The "ButtonRelease" event is the act of releasing the mouse button -- letting it up again.

We need to distinguish a mouse ButtonPress from a mouse ButtonRelease in order to support such features as "drag and drop", in which we do a ButtonPress on some GUI component, drag the component somewhere, and then "drop" it in the new location by by releasing the mouse button.

But button widgets are not the kind of thing that can be dragged-and-dropped. If a user thought that he could drag and drop a button, he might do a ButtonPress on the button widget, then drag the mouse pointer to someplace else on the screen, and release the mouse button. This is NOT the kind of activity that we want to recognize as a "push" (or -- technical term -- "invocation") of the button widget. For us to consider a button widget as having been pushed, we want the user to do a ButtonPress on the widget, and then -- WITHOUT moving the mouse pointer away from the widget -- to do a ButtonRelease. *THAT* is what we will consider to be a button press.

This is a more complicated notion of button invocation than we have used in our previous programs, where we simply bound a "Button-1" event to the button widget using event binding.

Fortunately, there is another form of binding that supports this more complicated notion of widget invocation. It is called "command binding" because it uses the widget's "command" option.

In this program, look at the lines with comments (1) and (2) to see how command binding is done. In those lines, we use the "command" option to bind button1 to the "self.button1Click" event handler, and to bind button2 to the "self.button2Click" event handler.

(3) (4)

Look at the definition of the event handlers themselves. Note that -- unlike the event handlers in the previous program -- they are NOT expecting an event object as an argument. That is because command binding, unlike event binding, does NOT automatically pass an event object as an argument. And of course, this makes sense. A command binding doesn't bind a single event to a handler. It binds multiple events to a handler. For a Button widget, for instance, it binds a pattern of a ButtonPress followed by a ButtonRelease to a handler. If it were to pass an event to its event handler, which event would it pass: ButtonPress or ButtonRelease? Neither would be exactly correct. This is why command binding, unlike event binding, does not pass an event object.

We will look a little bit more at this difference in our next program, but for now, let's run the program.

Program Behavior

When you run this program, the buttons that you see will look exactly like the buttons in the previous program... but they will behave differently.

Compare their behavior when doing a mouse ButtonPress on one of the buttons. For example, move the mouse pointer on the screen until it is positioned over the "OK" button widget, and then press down on the left mouse button BUT DO NOT LET THE MOUSE BUTTON UP.

If you do this with the previous example, the button1Click handler will be run immediately and you will see a message printed. But if you do this with this program, nothing will happen... UNTIL YOU RELEASE THE MOUSE BUTTON. When you release the mouse button, you will see a printed message.

There is another difference. Compare their behaviour when you do a keypress of the spacebar, and of the RETURN key. For example, use the TAB key to put the focus on the "OK" button, then press the spacebar or the RETURN key.

In the previous program (where we bound the "OK" button to the "Return" keypress event), pressing the spacebar has no effect but pressing the RETURN key causes the button to change color. In this program, on the other hand, the behavior is just the reverse -- pressing the spacebar causes the button to change color, but pressing the RETURN key has no effect.

We'll look at these behaviors in our next program.

[revised: 2002-10-01]

Program Source Code

```
from Tkinter import *

class MyApp:
    def __init__(self, parent):
        self.myParent = parent
        self.myContainer1 = Frame(parent)
        self.myContainer1.pack()

        self.button1 = Button(self.myContainer1, command=self.button1Click) ### (1)
        self.button1.configure(text="OK", background= "green")
        self.button1.pack(side=LEFT)
        self.button1.focus_force()


        self.button2 = Button(self.myContainer1, command=self.button2Click)  ### (2)
        self.button2.configure(text="Cancel", background="red")
        self.button2.pack(side=RIGHT)


    def button1Click(self):  ### (3)
        print "button1Click event handler"
        if self.button1["background"] == "green":
            self.button1["background"] = "yellow"
        else:
            self.button1["background"] = "green"

    def button2Click(self): ### (4)
```

```
        print "button2Click event handler"
        self.myParent.destroy()


root = Tk()
myapp = MyApp(root)
root.mainloop()
```

tt075.py

In the previous program, we introduced "command binding" and pointed out some of the differences between event binding and command binding. In this program, we explore those differences in a bit more detail.

What Events Does "command" Bind To?

In the previous program, if you use the TAB key to put the focus on the "OK" button, pressing the spacebar causes the button to change color, but pressing the RETURN key has no effect.

The reason for this is that the "command" option for a Button widget provides keyboard-event awareness as well as mouse-event awareness. The keyboard event that it listens for is a press of the spacebar, not of the RETURN key. So, with command binding, pressing the spacebar will cause the OK button to change color, but pressing the RETURN key will have no effect.

This behavior seems (to me, at least, with my Windows background) unusual. So part of the moral here is that if you are going to use command binding, it is a good idea to understand exactly what it is you are binding to. That is, it is a good idea to understand exactly what keyboard and/or mouse events cause the command to be invoked.

Unfortunately, the only really reliable source of this information is the Tk source code itself. For more accessible information, you can check books about Tk (Brent Welch's "Practical Programming in Tcl and Tk" is especially good) or about Tkinter. The Tk documentation is spotty, but available online. For version 8.4 of Tcl, the online documentation is available at:

```
http://www.tcl.tk/man/tcl8.4/TkCmd/contents.htm
```

You should also know that not all widgets provide a "command" option. Most of the various Button widgets (RadioButton, CheckButton, etc.) do. And others provide similar options (e.g. scrollcommand). But you really have to investigate each different kind of widget to find out whether it supports command binding. But by all means learn about the "command" option for the widgets that you will be using. It will improve the behavior of your GUI, and make your life as a coder easier.

Using Event Binding And Command Binding Together

We noted in our last program that command binding, unlike event binding, does NOT automatically pass an event object as an argument. This can make life a little complicated if you want to bind an event handler to a widget using *both* event binding *and* command binding.

For example, in this program we really would like our buttons to respond to presses of the RETURN key as well as the spacebar. But to get them to do that, we will have to use event binding to the <Return> keyboard event, like we did in our earlier program. (1)

The problem is that the command binding will not pass an event object as an argument, but the event binding will. So how should we write our event handler?

There are a number of solutions to this problem, but the simplest is to write two event handlers. The "real" event handler (2) will be the one used by the "command" binding, and it won't expect an event object.

The other event handler (3) will just be a wrapper for the real event-handler. This wrapper will expect the event-object argument, but will ignore it and call the real event-handler without it. We will give the wrapper the same name as the real event-handler, but with an added "_a" suffix.

Program Behavior

If you run this program, the behavior will be the same as the previous program, except for the fact that now the buttons will respond to a press of the RETURN key as well as the spacebar.

[revised: 2002-10-01]

Program Source Code

```
from Tkinter import *

class MyApp:
    def __init__(self, parent):
        self.myParent = parent
        self.myContainer1 = Frame(parent)
        self.myContainer1.pack()

        self.button1 = Button(self.myContainer1, command=self.button1Click)
        self.button1.bind("<Return>", self.button1Click_a)    ### (1)
        self.button1.configure(text="OK", background= "green")
        self.button1.pack(side=LEFT)
        self.button1.focus_force()

        self.button2 = Button(self.myContainer1, command=self.button2Click)
        self.button2.bind("<Return>", self.button2Click_a)    ### (1)
        self.button2.configure(text="Cancel", background="red")
        self.button2.pack(side=RIGHT)

    def button1Click(self):  ### (2)
        print "button1Click event handler"
        if self.button1["background"] == "green":
            self.button1["background"] = "yellow"
        else:
            self.button1["background"] = "green"

    def button2Click(self): ### (2)
        print "button2Click event handler"
        self.myParent.destroy()

    def button1Click_a(self, event):  ### (3)
        print "button1Click_a event handler (a wrapper)"
        self.button1Click()

    def button2Click_a(self, event):  ### (3)
        print "button2Click_a event handler (a wrapper)"
        self.button2Click()


root = Tk()
myapp = MyApp(root)
root.mainloop()
```

tt076.py

In the last few programs, we explored ways to make your programs actually do work with event handlers.

In this program, we take a quick look at sharing information between event handlers.

Sharing Information Between Event-handler Functions

There are a variety of situations in which you might want an event handler to perform some task, and then share the results of that task with other event handlers in your program.

A common pattern is that you have an application with two sets of widgets. One set of widgets sets up, or chooses, some piece of information, and the other set of widgets does something with the information.

For example, you might have a widget that allows a user to pick a filename from a list of filenames, and another set of widgets that offer various operations on the file that was picked -- opening the file, deleting it, copying it, renaming it, and so on.

Or you might have one set of widgets that sets various configuration options for your application, and another set (buttons offering SAVE and CANCEL options, perhaps) that either save those settings to disk, or cancel without saving them.

Or you might have one set of widgets that sets up parameters for a program that you want to run, and another widget (probably a button with a name like RUN or EXECUTE) that starts the program running with those parameters.

Or you might need one invocation of an event-handler function to pass information to a later invocation of the same function. Consider an event handler that simply toggles a variable back and forth between two different values. In order for it to assign a new value to the variable, it has to know what value it assigned to the variable the last time it was run.

The Problem

The problem here is that each event handler is a separate function. Each event handler has its own local variables that it does not share with other event-handler functions, or even with later invocations of itself. So the problem is this -- How can an event-handler function share data with other handlers, if it can't share its local variables with them?

The solution, of course, is that the variables that need to be shared cannot be local to the event handler functions. They must be stored *outside* of the event handler functions.

Solution 1 -- Use Global Variables

One technique for doing this is to make them (the variables that you want to share) global. For example, in each handler that needs to change or see myVariable1 and myVariable2, you can put the

statement:

```
global myVariable1, myVariable2
```

But the use of global variables is potentially dangerous, and is generally frowned upon as sloppy programming.

Solution 2 -- Use Instance Variables

A much cleaner technique is to use "instance variables" (that is, "self." variables) to hold the information that you want to share between event handlers. To do this, of course, your application must be implemented in a class, and not simply as in-line code.

This was one of the reasons that (earlier in this series) we put our application into a class. Since we did that earlier, at this point our application already has an infrastructure that will allow us to use instance variables.

In this program, we are going to remember and share a very simple piece of information: the name of the last button that was invoked. We will store it in an instance variable called "self.myLastButtonInvoked". [see ### 1 comments]

And to show that we actually are remembering this information, whenever a button handler is invoked, it will print out this information. [see ### 2 comments]

Program Behavior

This program shows three buttons. When you run this program, if you click on any of the buttons, it will display its own name, and the name of the previous button that was clicked.

Note that none of the buttons will close the application, so when you are ready to close it, you must use the CLOSE widget (the icon with an "X" in a box, at the right side of the titlebar).

[revised: 2002-10-05]

Program Source Code

```
from Tkinter import *

class MyApp:
    def __init__(self, parent):

        ### 1 -- At the outset, we haven't yet invoked any button handler.
        self.myLastButtonInvoked = None

        self.myParent = parent
        self.myContainer1 = Frame(parent)
        self.myContainer1.pack()

        self.yellowButton = Button(self.myContainer1, command=self.yellowButtonClick)
        self.yellowButton.configure(text="YELLOW", background="yellow")
        self.yellowButton.pack(side=LEFT)

        self.redButton = Button(self.myContainer1, command=self.redButtonClick)
        self.redButton.configure(text="RED", background= "red")
        self.redButton.pack(side=LEFT)
```

```
        self.whiteButton = Button(self.myContainer1, command=self.whiteButtonClick)
        self.whiteButton.configure(text="WHITE", background="white")
        self.whiteButton.pack(side=LEFT)

    def redButtonClick(self):
        print "RED    button clicked.  Previous button invoked was", self.myLastButtonInvoked
        self.myLastButtonInvoked = "RED"  ### 1

    def yellowButtonClick(self):
        print "YELLOW button clicked.  Previous button invoked was", self.myLastButtonInvoked
        self.myLastButtonInvoked = "YELLOW" ### 1

    def whiteButtonClick(self):
        print "WHITE  button clicked.  Previous button invoked was", self.myLastButtonInvoked
        self.myLastButtonInvoked = "WHITE" ### 1


print "\n"*100 # a simple way to clear the screen
print "Starting..."
root = Tk()
myapp = MyApp(root)
root.mainloop()
print "... Done!"
```

tt077.py

In this program we explore some ...

More Advanced Features Of Command Binding

In our program tt075.py, we used the "command" option to bind an event handler to a widget. For example, in that program the statement

```
self.button1 = Button(self.myContainer1, command=self.button1Click)
```

bound the button1Click function to the button1 widget.

And we used event binding to bind our buttons to the <Return> keyboard event.

```
self.button1.bind("", self.button1Click_a)
```

In our earlier program, the event handlers for the two buttons performed quite different functions.

But suppose that the situation was different. Suppose that we have several buttons, all of which should trigger essentially the *same* type of action. The best way to handle such a situation is to bind the events for all of the buttons to a single event handler. Each button would invoke the same handler routine, but pass it different arguments telling it what to do.

That is what we are doing in this program.

Command Binding

In this program, as you can see, we have two buttons, and we use the "command" option to bind them all to the same event handler -- the "buttonHandler" routine. We pass the buttonHandler routine three arguments: the name of the button (in the button_name variable), a number, and a string.

```
self.button1 = Button(self.myContainer1,
    command=self.buttonHandler(button_name, 1, "Good stuff!")
    )
```

In a serious application, the buttonHandler routine would of course do serious work, but in this program it merely prints the arguments that it receives.

Event Binding

So much for command binding. What about event binding?

You will note that we have commented out the two lines that do event binding on the <Return> event.

```
# self.button1.bind("", self.buttonHandler_a(event, button_name, 1, "Good stuff!"))
```

This is the first sign of a problem. Event binding automatically passes an event argument -- but there is simply no way to include that event argument in our list of arguments.

We'll have to come back to this problem later. For now, let's simply run the program and see what happens.

Program Behavior

When you look at the code, this program looks quite reasonable. But when you run it, you will see that it doesn't work right. The buttonHandler routine is invoked even before the GUI is displayed. In fact, it is invoked TWO times!

And if you left-mouse-click on any of the buttons, you will find that nothing happens -- the "eventHandler" routine is *not* being invoked.

Note that the only way to close this program is to click the "close" icon (the "X" in a box) on the right side of the title bar.

So run the program now, and see what happens. Then, in our next program, we will see why it happens.

[revised: 2003-02-23]

Program Source Code

```
from Tkinter import *

class MyApp:
    def __init__(self, parent):
        self.myParent = parent
        self.myContainer1 = Frame(parent)
        self.myContainer1.pack()

        button_name = "OK"
        self.button1 = Button(self.myContainer1,
            command=self.buttonHandler(button_name, 1, "Good stuff!"))

        # self.button1.bind("<Return>", self.buttonHandler_a(event, button_name, 1, "Good stuf
        self.button1.configure(text=button_name, background="green")
        self.button1.pack(side=LEFT)
        self.button1.focus_force()  # Put keyboard focus on button1

        button_name = "Cancel"
        self.button2 = Button(self.myContainer1,
            command=self.buttonHandler(button_name, 2, "Bad  stuff!"))

        # self.button2.bind("<Return>", self.buttonHandler_a(event, button_name, 2, "Bad  stuf
        self.button2.configure(text=button_name, background="red")
        self.button2.pack(side=LEFT)


    def buttonHandler(self, arg1, arg2, arg3):
        print "    buttonHandler routine received arguments:", arg1.ljust(8), arg2, arg3

        def buttonHandler_a(self, event, arg1, arg2, arg3):
        print "buttonHandler_a received event", event
        self.buttonHandler(arg1, arg2, arg3)

print "\n"*100 # clear the screen
print "Starting program tt077."
root = Tk()
myapp = MyApp(root)
```

```
print "Ready to start executing the event loop."
root.mainloop()
print "Finished       executing the event loop."
```

tt078.py

Looking at the execution of the last program, we have to ask: "What's happening here??!! The "buttonHandler" routine is being executed for each button, even before the event loop is started!!"

The reason is that in a statement like

```
  self.button1 = Button(self.myContainer1,
       command = self.buttonHandler(button_name, 1, "Good stuff!"))
```

we are calling the buttonHandler function, rather than asking that it be used as a callback. That's not what we intended to do, but that is what we're actually doing.

Note That -->

```
 * buttonHandler is a function object, and can be used as a callback binding.

 * buttonHandler() (note the parenthesis) on the other hand is an actual call to
   the "buttonHandler" function.
```

At the time that the statement

```
  self.button1 = Button(self.myContainer1,
       command = self.buttonHandler(button_name, 1, "Good stuff!"))
```

is executed, it is actually making a call to the "buttonHandler" routine. The buttonHandler routine executes, printing its message, and returns the results of the call (in this case, the None object). Then the button's "command" option is bound to the results of the call. In short, the command is bound to the "None" object. And that is why, when you click on either of the buttons, nothing happens.

Is There A Solution?

So... what's the solution? Is there any way to parameterize, and reuse, an event-handler function?

Yes. There are a couple of generally recognized techniques for doing this. One uses the Python built-in "lambda" function, and the other uses a technique called "currying".

In this program we will discuss how to work with lambda, and in the next program we will look at currying.

I'm not going to try to explain how lambda and currying work -- it is too complicated and too far off-track from our main goal, which is to get Tkinter programs working. So I'm going to simply treat them as black boxes. I won't talk about how they work -- I'll only talk about how to work with them.

So let's look at lambda.

Command Binding

Originally, we thought the following statement might work:

```
self.button1 = Button(self.myContainer1,
      command = self.buttonHandler(button_name, 1, "Good stuff!")
      )
```

... but we found out that it didn't work the way we thought it would.

The way to do what we want is to re-write this statement this way:

```
self.button1 = Button(self.myContainer1,
     command = lambda
     arg1=button_name, arg2=1, arg3="Good stuff!" :
     self.buttonHandler(arg1, arg2, arg3)
     )
```

Event Binding

Happily, lambda also gives us a way to parameterize event binding. Instead of:

```
    self.button1.bind("",
       self.buttonHandler_a(event, button_name, 1, "Good stuff!"))
```

(which wouldn't work, because there was no way to include the event argument in the argument list), we can use lambda, this way:

```
# event binding -- passing the event as an argument
self.button1.bind("",
        lambda
        event, arg1=button_name, arg2=1, arg3="Good stuff!" :
        self.buttonHandler_a(event, arg1, arg2, arg3)
        )
```

[Note that "event" here is a variable name -- it is not a Python keyword or anything like that. This example uses the name "event" for the event argument, but some discussions of this technique use the name "e" for the event argument, and we could just as easily have called it "event_arg" if we had wanted to.]

One of the nice features of using lambda is that we can (if we wish), simply not pass the event argument. If we don't pass the event argument, then we can call the self.buttonHandler function directly, instead of indirectly through the self.buttonHandler_a function.

To illustrate this technique, we will code the event binding for button2 differently than we did for button1. This is what we do with button2:

```
# event binding -- without passing the event as an argument
self.button2.bind("",
        lambda
        event, arg1=button_name, arg2=2, arg3="Bad  stuff!" :
        self.buttonHandler(arg1, arg2, arg3)
        )
```

Program Behavior

If you run this program, it will behave just as we wish.

Note that you can change the keyboard focus from the OK to the CANCEL button, and back again, by pressing the TAB key on the keyboard.

In particular, you should experiment with invoking the OK button by pressing the <Return> key on the keyboard. If you invoke the OK button via a keypress of the <Return> key, you will be going through the buttonHandler_a function, and you will also get a message from it, printing information about the event that has been passed to it.

In any case, whether you click on one of the button widgets with the mouse, or invoke a widget via a <Return> keypress on the keyboard, it will nicely print the arguments that were passed to the buttonHandler function.

[revised: 2003-02-23]

Program Source Code

```
from Tkinter import *

class MyApp:
    def __init__(self, parent):
        self.myParent = parent
        self.myContainer1 = Frame(parent)
        self.myContainer1.pack()

        #------------------ BUTTON #1 -----------------------------------
        button_name = "OK"

        # command binding
        self.button1 = Button(self.myContainer1,
            command = lambda
            arg1=button_name, arg2=1, arg3="Good stuff!" :
            self.buttonHandler(arg1, arg2, arg3)
            )

        # event binding -- passing the event as an argument
        self.button1.bind("<Return>",
            lambda
            event, arg1=button_name, arg2=1, arg3="Good stuff!" :
            self.buttonHandler_a(event, arg1, arg2, arg3)
            )

        self.button1.configure(text=button_name, background="green")
        self.button1.pack(side=LEFT)
        self.button1.focus_force()  # Put keyboard focus on button1

        #------------------ BUTTON #2 -----------------------------------
        button_name = "Cancel"

        # command binding
        self.button2 = Button(self.myContainer1,
            command = lambda
            arg1=button_name, arg2=2, arg3="Bad  stuff!":
            self.buttonHandler(arg1, arg2, arg3)
            )

        # event binding -- without passing the event as an argument
        self.button2.bind("<Return>",
            lambda
            event, arg1=button_name, arg2=2, arg3="Bad  stuff!" :
            self.buttonHandler(arg1, arg2, arg3)
            )

        self.button2.configure(text=button_name, background="red")
        self.button2.pack(side=LEFT)
```

```
    def buttonHandler(self, argument1, argument2, argument3):
        print "    buttonHandler routine received arguments:" \
            , argument1.ljust(8), argument2, argument3

        def buttonHandler_a(self, event, argument1, argument2, argument3):
        print "buttonHandler_a received event", event
        self.buttonHandler(argument1, argument2, argument3)


print "\n"*100 # clear the screen
print "Starting program tt078."
root = Tk()
myapp = MyApp(root)
print "Ready to start executing the event loop."
root.mainloop()
print "Finished       executing the event loop."
```

tt079.py

In the previous program, we looked at a technique involving lambda to pass arguments to an event-handler function. In this program, we will look at a different technique, called "currying".

About Curry

In its very simplest sense, currying is a technique for using a function to construct other functions.

Currying is a technique borrowed from functional programming. If you would like to learn more about currying, there are several recipes available in the "Python Cookbook":

```
http://aspn.activestate.com/ASPN/Python/Cookbook/
```

The curry class used in this program is from Scott David Daniels' recipe "curry -- associating parameters with a function", available at http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52549

As in our discussion of lambda, I'm not going to try to explain how currying works. I'm simply going to treat the curry class as a black box. I won't talk much about how it works -- I'll only talk about how to work with it.

Curry -- How To Use It

The way to use curry (the technique) is to include a "curry" class in your program, or to import it from its own Python file. In this program, we will include the curry code directly in the program.

Originally, we thought that the following statement could bind self.buttonHandler to self.button1's command option, but we found out that it didn't work the way we thought it would.

```
self.button1 = Button(self.myContainer1,
     command = self.buttonHandler(button_name, 1, "Good stuff!"))
```

Using curry, the way to do what we want is to re-write this statement this way:

```
self.button1 = Button(self.myContainer1,
     command = curry(self.buttonHandler, button_name, 1, "Good stuff!"))
```

As you can see, the code is quite straightforward. Instead of invoking the self.buttonHandler function, we create a curry object (that is, an instance of the curry class), passing the self.buttonHandler function in as the first argument. Basically, what happens is that the curry object remembers the name of the function that it was given. Then when it (the curry object) is called, it -- in its own turn -- calls the function that it was given when it was created.

Event Binding

Chad Netzer has devised a technique similar to currying, that can be used to parameterize event binding. [NOTE that this coding of the technique requires Python 2 or greater.] It involves using an "event_lambda" function.

To use event_lambda, as with curry, you need to include the code for the "event_lambda" function in your program, or to import it from its own Python file. In this program, we include the code for the event_lambda function directly in our program.

```
# ---------- code for function: event_lambda --------
def event_lambda(f, *args, **kwds ):
        "A helper function that wraps lambda in a prettier interface."
        return lambda event, f=f, args=args, kwds=kwds : f( *args, **kwds )
```

Once the event_lambda function is available to us, we can use it to bind self.buttonHandler to the <Return> keyboard event, and pass it some arguments. Here's how we do it:

```
self.button1.bind("",
        event_lambda( self.buttonHandler, button_name, 1, "Good stuff!" ) )
```

If you have an absolutely insatiable curiosity about how event_lambda works, it is a little bit easier to see by looking at the coding for button2.

For button2, use use a two-step process. First we invoke the event_lambda function.

```
event_handler = event_lambda( self.buttonHandler, button_name, 2, "Bad  stuff!" )
```

When the event_lambda function is called, it uses lambda to create a new, un-named ("anonymous") function object.

```
        lambda event, f=f, args=args, kwds=kwds : f( *args, **kwds )
```

The un-named function object is a wrapper for the function we really want to invoke ("f", which in this program is "self.buttonHandler") and the arguments we specified at the time we called the event_lambda function.

Then the event_lambda function returns this new anonymous function.

When event_lambda returns the anonymous function, we give it a name: "event_handler".

```
event_handler = event_lambda( self.buttonHandler, button_name, 2, "Bad  stuff!" )
```

Then, in the second step, we bind the <Return> event to the "event_handler" function.

```
self.button2.bind("", event_handler )
```

Note that for the anonymous function, 'event' is just a placeholder argument which is discarded and never used. Only the positional arguments (args) and the keyword arguments (kwds) are passed to the button-handler routine.

Wow! I Just Burned Out A Bunch Of Brain Cells!!

This is tricky stuff. But don't think that you need to burn out a bunch of brain cells trying to understand how it all works. You don't need to know HOW "curry" and "event_lambda" work in order to use them. Just treat them like black boxes... just use them and don't worry about how they work.

Lambda Vs. Curry And Event_lambda -- Which Should I Use?

Well...

```
 * Code to invoke curry and event_lambda is relatively intuitive, short and simple.
   The downside is that using them requires you to include them in the code for
   your program, or to import them.

 * Lambda, in contrast, is built into the Python language -- you don't need to
   do anything special to import it; it is simply there.
   The downside is that the code to use it can be long and a bit confusing.
```

So the choice is up to you. "You pays yer money and takes yer choice," as they say. Use what you are most comfortable with, and/or what seems most appropriate to the task.

The REAL moral of this story is this...

Python is a powerful language, and it provides many tools that can be used to create callback functions for handling events. "Thinking in Tkinter" is an introduction to basic concepts, not an encyclopedia of techniques, so we can explore only a few of those ways here. But you can be confident that as you become more skilled with Python, and as your need for more flexibility grows, there are more advanced features of Python that will be available to you, and that can enable you to create just the kind of callback function that you need.

Program Behavior

If you run this program, it will behave exactly as the previous program did. We haven't changed any of the behavior of the program, just the way the program is coded.

[revised: 2003-02-23]

Program Source Code

```
from Tkinter import *

# ---------- code for class: curry (begin) ---------------------
class curry:
    """from Scott David Daniels'recipe
    "curry -- associating parameters with a function"
    in the "Python Cookbook"
    http://aspn.activestate.com/ASPN/Python/Cookbook/
    """

    def __init__(self, fun, *args, **kwargs):
        self.fun = fun
        self.pending = args[:]
        self.kwargs = kwargs.copy()

    def __call__(self, *args, **kwargs):
        if kwargs and self.kwargs:
            kw = self.kwargs.copy()
            kw.update(kwargs)
        else:
            kw = kwargs or self.kwargs
        return self.fun(*(self.pending + args), **kw)
# ---------- code for class: curry (end) ---------------------


# ---------- code for function: event_lambda (begin) --------
def event_lambda(f, *args, **kwds ):
    """A helper function that wraps lambda in a prettier interface.
    Thanks to Chad Netzer for the code."""
    return lambda event, f=f, args=args, kwds=kwds : f( *args, **kwds )
```

```
# ---------- code for function: event_lambda (end) -----------


class MyApp:
    def __init__(self, parent):
        self.myParent = parent
        self.myContainer1 = Frame(parent)
        self.myContainer1.pack()

        button_name = "OK"

        # command binding -- using curry
        self.button1 = Button(self.myContainer1,
           command = curry(self.buttonHandler, button_name, 1, "Good stuff!"))

        # event binding -- using the event_lambda helper function
        self.button1.bind("<Return>",
            event_lambda( self.buttonHandler, button_name, 1, "Good stuff!" ) )

        self.button1.configure(text=button_name, background="green")
        self.button1.pack(side=LEFT)
        self.button1.focus_force()  # Put keyboard focus on button1


        button_name = "Cancel"

        # command binding -- using curry
        self.button2 = Button(self.myContainer1,
            command = curry(self.buttonHandler, button_name, 2, "Bad  stuff!"))

        # event binding -- using the event_lambda helper function in two steps
        event_handler = event_lambda( self.buttonHandler, button_name, 2, "Bad  stuff!" )
        self.button2.bind("<Return>", event_handler )

        self.button2.configure(text=button_name, background="red")
        self.button2.pack(side=LEFT)


    def buttonHandler(self, argument1, argument2, argument3):
        print "    buttonHandler routine received arguments:", \
            argument1.ljust(8), argument2, argument3

    def buttonHandler_a(self, event, argument1, argument2, argument3):
        print "buttonHandler_a received event", event
        self.buttonHandler(argument1, argument2, argument3)

print "\n"*100 # clear the screen
print "Starting program tt079."
root = Tk()
myapp = MyApp(root)
print "Ready to start executing the event loop."
root.mainloop()
print "Finished       executing the event loop."
```

tt080.py

In the last several programs, we've spent a lot of time discussing techniques for binding event handlers to widgets.

With this program, we return to the topic of creating the appearance of the GUI -- setting up the widgets and controlling their appearance and location.

Three Techniques For Controlling The Layout Of A Gui

There are three techniques for controlling the general layout of a GUI.

```
* widget attributes
* pack() options
* nesting of containers (frames)
```

In this program, we look at controlling appearance through setting of widget attributes and pack() options.

We're going to be working a lot with the buttons, and with the frame that contains them. In earlier versions of this program, we called the frame "myContainer1". Here, we have renamed it something a bit more descriptive: "buttons_frame".

The numbers of the following section refer to the numbered comments in the source code.

(1) First, to make sure that all of our buttons are the same width, we specify a "width" attribute that is the same for all of them. Note that the "width" attribute is specific to Tkinter's "Button" widget -- not all widgets have a width attribute. Note also that the width attribute is specified in units of characters (not, for example, in units of pixels, inches, or millimeters). Since our longest label ("Cancel") contains six characters, we set the width for our buttons to "6". (1)

(2) Now we add some padding to our buttons. Padding is extra space that goes around the text, between the text and the edge of the button. We do this by setting the "padx" and "pady" attributes of the buttons. "padx" pads along the X-axis, horizontally, to the left and right. "pady" pads along the Y-axis, vertically, to the top and bottom.

We will specify our horizontal padding as 3 millimeters (padx="3m") and our vertical padding as 1 millimeter (pady="1m"). Note that, unlike the "width" attribute (which was numeric), these attributes are enclosed in quotation marks. That is because we are specifying the units of padding with the "m" suffix, so we must specify the padding lengths as strings rather than as numbers.

(3) Finally, we add some padding to the container (buttons_frame) that holds the buttons. For the container, we can specify four padding attributes. "padx" and "pady" specify the padding that goes around (outside) the frame. "ipadx" and "ipady" ("internal padx" and "internal pady") specify the internal padding. This is padding that goes around each of the widgets that are inside the container.

Notice that we don't specify the padding for the frame as an attribute of the frame, but as options that

we pass to the packer. (4). As you can see, the padding is a bit confusing. Frames have internal padding, but widget such as buttons do not. In some cases, the padding is an attribute of the widget, while in other cases it is specified as an option to pack().

Program Behavior

When you run this program, you will see two buttons. But now there they should both be the same size. The sides of the buttons don't grip the button-text so closely. And the buttons are surrounded by a nice border of space.

[revised: 2003-02-23

Program Source Code

```
from Tkinter import *

class MyApp:
    def __init__(self, parent):

        #------ constants for controlling layout ------
        button_width = 6      ### (1)

        button_padx = "2m"    ### (2)
        button_pady = "1m"    ### (2)

        buttons_frame_padx =  "3m"   ### (3)
        buttons_frame_pady =  "2m"   ### (3)
        buttons_frame_ipadx = "3m"   ### (3)
        buttons_frame_ipady = "1m"   ### (3)
        # -------------- end constants ----------------

        self.myParent = parent
        self.buttons_frame = Frame(parent)

        self.buttons_frame.pack(    ### (4)
            ipadx=buttons_frame_ipadx,  ### (3)
            ipady=buttons_frame_ipady,  ### (3)
            padx=buttons_frame_padx,    ### (3)
            pady=buttons_frame_pady,    ### (3)
            )


        self.button1 = Button(self.buttons_frame, command=self.button1Click)
        self.button1.configure(text="OK", background= "green")
        self.button1.focus_force()
        self.button1.configure(
            width=button_width,  ### (1)
            padx=button_padx,    ### (2)
            pady=button_pady     ### (2)
            )

        self.button1.pack(side=LEFT)
        self.button1.bind("<Return>", self.button1Click_a)

        self.button2 = Button(self.buttons_frame, command=self.button2Click)
        self.button2.configure(text="Cancel", background="red")
        self.button2.configure(
            width=button_width,  ### (1)
            padx=button_padx,    ### (2)
            pady=button_pady     ### (2)
```

```
            )

        self.button2.pack(side=RIGHT)
        self.button2.bind("<Return>", self.button2Click_a)

    def button1Click(self):
        if self.button1["background"] == "green":
            self.button1["background"] = "yellow"
        else:
            self.button1["background"] = "green"

    def button2Click(self):
        self.myParent.destroy()

    def button1Click_a(self, event):
        self.button1Click()

    def button2Click_a(self, event):
        self.button2Click()



root = Tk()
myapp = MyApp(root)
root.mainloop()
```

tt090.py

In this program, we look at nesting of containers (frames). What we are going to do is to create a series of frames, nested inside each other: bottom_frame, left_frame, and big_frame.

These frames will contain nothing -- no widgets. Normally, because frames are elastic, the would shrink down to nothing. But we specify "height" and "width" attributes to provide an initial size for them.

Note that we don't provide heights or widths for all of the frames. For myContainer1, for instance, we don't provide either. But we do provide heights and widths for its descendants, and it will stretch to accommodate the cumulative heights and widths of its descendants.

In a later program we will explore putting widgets into these frames, but in this program we will simply create the frames, and give them different different sizes, positions, and background colors.

We also put a ridged border around the three frames that will be of most iterest to us in the future: bottom_frame, left_frame, and right_frame. The other frames (for instance top_frame and buttons_frame) are not given a border.

Program Behavior

When you run this program, you will see the different frames, with different background colors.

[revised: 2005-07-15]

Program Source Code

```
from Tkinter import *

class MyApp:
    def __init__(self, parent):

        self.myParent = parent

        ### Our topmost frame is called myContainer1
        self.myContainer1 = Frame(parent) ###
        self.myContainer1.pack()

        #------ constants for controlling layout ------
        button_width = 6      ### (1)

        button_padx = "2m"    ### (2)
        button_pady = "1m"    ### (2)

        buttons_frame_padx =  "3m"   ### (3)
        buttons_frame_pady =  "2m"   ### (3)
        buttons_frame_ipadx = "3m"   ### (3)
        buttons_frame_ipady = "1m"   ### (3)
        # -------------- end constants ----------------

        ### We will use VERTICAL (top/bottom) orientation inside myContainer1.
```

```
### Inside myContainer1, first we create buttons_frame.
### Then we create top_frame and bottom_frame.
### These will be our demonstration frames.

# buttons frame
self.buttons_frame = Frame(self.myContainer1) ###
self.buttons_frame.pack(
    side=TOP,   ###
    ipadx=buttons_frame_ipadx,
    ipady=buttons_frame_ipady,
    padx=buttons_frame_padx,
    pady=buttons_frame_pady,
    )

# top frame
self.top_frame = Frame(self.myContainer1)
self.top_frame.pack(side=TOP,
    fill=BOTH,
    expand=YES,
    )  ###

# bottom frame
self.bottom_frame = Frame(self.myContainer1,
    borderwidth=5,  relief=RIDGE,
    height=50,
    background="white",
    ) ###
self.bottom_frame.pack(side=TOP,
    fill=BOTH,
    expand=YES,
    )  ###


### Now we will put two more frames, left_frame and right_frame,
### inside top_frame.  We will use HORIZONTAL (left/right)
### orientation within top_frame.

# left_frame
self.left_frame = Frame(self.top_frame, background="red",
    borderwidth=5,  relief=RIDGE,
    height=250,
    width=50,
    ) ###
self.left_frame.pack(side=LEFT,
    fill=BOTH,
    expand=YES,
    )  ###


### right_frame
self.right_frame = Frame(self.top_frame, background="tan",
    borderwidth=5,  relief=RIDGE,
    width=250,
    )
self.right_frame.pack(side=RIGHT,
    fill=BOTH,
    expand=YES,
    )  ###


# now we add the buttons to the buttons_frame
self.button1 = Button(self.buttons_frame, command=self.button1Click)
self.button1.configure(text="OK", background= "green")
self.button1.focus_force()
self.button1.configure(
```

```
            width=button_width,  ### (1)
            padx=button_padx,    ### (2)
            pady=button_pady     ### (2)
            )

        self.button1.pack(side=LEFT)
        self.button1.bind("<Return>", self.button1Click_a)

        self.button2 = Button(self.buttons_frame, command=self.button2Click)
        self.button2.configure(text="Cancel", background="red")
        self.button2.configure(
            width=button_width,  ### (1)
            padx=button_padx,    ### (2)
            pady=button_pady     ### (2)
            )

        self.button2.pack(side=RIGHT)
        self.button2.bind("<Return>", self.button2Click_a)

    def button1Click(self):
        if self.button1["background"] == "green":
            self.button1["background"] = "yellow"
        else:
            self.button1["background"] = "green"

    def button2Click(self):
        self.myParent.destroy()

    def button1Click_a(self, event):
        self.button1Click()

    def button2Click_a(self, event):
        self.button2Click()


root = Tk()
myapp = MyApp(root)
root.mainloop()
```

tt095.py

Sizing windows can be a frustrating experience when working with Tkinter. Imagine this situation. You believe in iterative development, so first you carefully lay out a frame with the height and width specification that you want. You test it and you see that it works. Then you move on to the next step, and add some buttons to the frame. You test it again, but now to your surprise Tkinter is acting as if there were no "height" and "width" specifications for the frame, and the frame has snapped down to tightly encase the buttons.

What's going on ???!!!

Well, the packer's behavior is inconsistent. Or, shall we say: the packer's behavior depends on a lot of situational factors. The bottom line is that the packer will honor the size request of a container if the container is empty, but if the container contains any other widgets, then the elastic nature of the container comes to the fore -- the "height" and "width" settings for the container are ignored, and the size of the container is adjusted to enclose the widgets as closely as possible.

The bottom line is that you really cannot control the size of a container that contains widgets.

What you CAN control is the the initial size of the entire root window, and you do this with the Window Manager "geometry" option.

(1)

In this program, we use the geometry option to make a nice big window around our smaller frame.

(2) Note that the "title" option, which we also use in this program, is also a Window Manager method. "Title" controls the text of the title in the window's title bar.

Not also that Window Manager options can optionally be specified with a "wm_" prefix, e.g. "wm_geometry" and "wm_title". In this program, just to show that it can be done either way, we use "geometry" and "wm_title".

Program Behavior

This program puts up four windows in succession.

Note that you will have to close each window by clicking on the "close" widget -- the "X" in a box at the right of the title bar.

In case 1, we see what a frame looks like when height and width are specified, and -- NOTE -- it contains no widgets.

In case 2, we see what the exact same frame looks like when some widgets (in our case, three buttons) are added to it. Note that the frame has snapped down tight around the three buttons.

In case 3, we again show what the empty frame looks like, only this time we use the geometry option

to control the size of the window as a whole. We can see the blue background of the frame inside the larger gray field of the window.

In case 4, we again show the frame with three buttons in it, but this time specifying the frame size with the geometry option. Note that the size of the window is the same as it was in Case 3, but (as in Case 2) the frame has snapped down around the buttons, and we cannot see any of the frame's blue background at all.

[revised: 2002-10-01]

Program Source Code

```
from Tkinter import *

class App:
    def __init__(self, root, use_geometry, show_buttons):
        fm = Frame(root, width=300, height=200, bg="blue")
        fm.pack(side=TOP, expand=NO, fill=NONE)

        if use_geometry:
            root.geometry("600x400")  ### (1) Note geometry Window Manager method

        if show_buttons:
            Button(fm, text="Button 1", width=10).pack(side=LEFT)
            Button(fm, text="Button 2", width=10).pack(side=LEFT)
            Button(fm, text="Button 3", width=10).pack(side=LEFT)


case = 0
for use_geometry in (0, 1):
    for show_buttons in (0,1):
        case = case + 1
        root = Tk()
        root.wm_title("Case " + str(case))  ### (2) Note wm_title Window Manager method
        display = App(root, use_geometry, show_buttons)
        root.mainloop()
```

tt100.py

In this program, we look at several pack() options for controlling layouts within a frame:

```
* side
* fill
* expand
* anchor
```

This program is unlike the other programs in this series. That is, you don't need to read the source code in order to understand how to code some feature. You need to RUN the program.

The purpose of the program is to show you the results of pack options. Running the program will allow you to set different pack options and to observe the effects of different combinations of options.

The Concepts Underlying The Pack Options

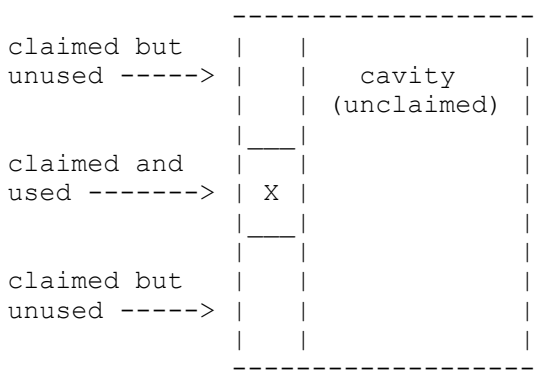
To understand how we can control the appearance of widgets within a container (that is, with a frame), we need to remember that the pack geometry manager uses a "cavity" model of arrangement. That is, each container contains a cavity, and we pack slaves within the container.

In talking about the positioning and display of components with a container, it will be useful to have three concepts:

```
* unclaimed space (that is, the cavity)
* claimed but unused space
* claimed and used  space
```

When you pack a widget, such as a button, it is always packed along one of the four sides of the cavity. The pack "side" option specifies which side to use. For example, if we specify "side=LEFT", the the widget will be packed (that is, positioned) on the left side of the cavity.

When a widget is packed along a side, it claims the entire side, although it may not actually use all of the space it has claimed. Suppose we pack a small button called X along the left side of a large cavity, as in the following diagram.



The cavity (the unclaimed area) is now to the right of the widget. Widget X has claimed the entire left

side, in a strip that is wide enough to hold it. But because widget X is small, it actually uses only a small part of the area that it has claimed. That is the part that it uses to display itself.

As you can see, widget X has claimed only enough space as it needs to display itself. If we specify a pack option of "expand=YES", it will claim ALL of the available area. No part of the cavity would be left unclaimed. Note that this does not mean that widget X would *use* the whole area. It would still only use the small part that it needs.

If a widget has claimed more space than it is using, it has two choices:

```
 * it can move around in the unclaimed space, or
 * it can grow to fill the unclaimed space.
```

If we want it to grow to use the the unclaimed space, we can use the "fill" option, which tells a widget whether or not it can grow to fill the unused space, and in which direction it can grow.

```
  * "fill=NONE" means that it cannot grow.
  * "fill=X"    means that it can grow along the X-axis (i.e. horizontally).
  * "fill=Y"    means that it can grow along the Y-axis (i.e. vertically).
  * "fill=BOTH" means that it can grow both horizontally and vertically.
```

If we want it to grow to move around in the the unclaimed space, then we can use the "anchor" option, which tells a widget where to position itself in the space it has claimed. The values of the anchor option are like compass headings. "N" means "north" (i.e. centered at the top of the claimed area). "NE" means "northeast" (i.e. in the upper, right corner of the claimed area), "CENTER" means centered right in the middle of the claimed area. And so on.

Running The Program

So now, run the program. You don't need to read the code. Just run the program and experiment with the various pack options of the three demo buttons.

Button A's frame gives it a horizontal cavity to run around in -- the frame is no taller than the button.

Button B's cavity gives it a vertical cavity to run around in -- the frame is no wider than the button.

And button C's frame gives it a big cavity -- much wider and taller than the button itself -- to play in.

If the appearance of any of the buttons under certain settings surprises you, try to figure out why the button looks the way it does.

And finally....

A Useful Debugging Tip

Note that packing is a complicated business, because the positioning of a widget with respect to other widgets that were packed earlier depends in part on how the other widgets were packed. That is, if the other widgets were packed to the left, then the cavity within which the next widget can be packed will be to their right. But if they were packed to the top of the cavity, then the cavity within which the next widget can be packed will be below them. It can all get very confusing.

Here is a useful debugging tip. If you are developing your layout and hit a problem -- things aren't acting the way you think they should -- then give each of your containers (that is, each of your frames)

a different background color, for example:

```
bg="red"
```

or

```
bg="cyan"
```

or

```
bg="tan"
```

... or yellow, or blue, or red, and so on.

This will allow you to see how the frames are actually arranging themselves. Often, what you see will give you a clue as to what the problem is.

[revised: 2004-04-26]

Program Source Code

```
from Tkinter import *

class MyApp:
    def __init__(self, parent):

        #------ constants for controlling layout of buttons ------
        button_width = 6
        button_padx = "2m"
        button_pady = "1m"
        buttons_frame_padx =  "3m"
        buttons_frame_pady =  "2m"
        buttons_frame_ipadx = "3m"
        buttons_frame_ipady = "1m"
        # -------------- end constants ----------------


        # set up Tkinter variables, to be controlled by the radio buttons
        self.button_name   = StringVar()
        self.button_name.set("C")

        self.side_option = StringVar()
        self.side_option.set(LEFT)

        self.fill_option   = StringVar()
        self.fill_option.set(NONE)

        self.expand_option = StringVar()
        self.expand_option.set(YES)

        self.anchor_option = StringVar()
        self.anchor_option.set(CENTER)


        # -------------- end constants ----------------

        self.myParent = parent
        self.myParent.geometry("640x400")

        ### Our topmost frame is called myContainer1
```

```
self.myContainer1 = Frame(parent) ###
self.myContainer1.pack(expand=YES, fill=BOTH)


### We will use HORIZONTAL (left/right) orientation inside myContainer1.
### Inside myContainer1, we create control_frame and demo_frame.

# control frame - basically everything except the demo frame
self.control_frame = Frame(self.myContainer1) ###
self.control_frame.pack(side=LEFT, expand=NO,  padx=10, pady=5, ipadx=5, ipady=5)

# inside control_frame we create a header label
# and a buttons_frame at the top,
# and demo_frame at the bottom

myMessage="This window shows the effects of the \nexpand, fill, and anchor packing opt
Label(self.control_frame, text=myMessage, justify=LEFT).pack(side=TOP, anchor=W)

# buttons frame
self.buttons_frame = Frame(self.control_frame) ###
self.buttons_frame.pack(side=TOP, expand=NO, fill=Y, ipadx=5, ipady=5)

# demo frame
self.demo_frame = Frame(self.myContainer1) ###
self.demo_frame.pack(side=RIGHT, expand=YES, fill=BOTH)


### Inside the demo frame, we create top_frame and bottom_frame.
### These will be our demonstration frames.
# top frame
self.top_frame = Frame(self.demo_frame)
self.top_frame.pack(side=TOP, expand=YES, fill=BOTH)  ###

# bottom frame
self.bottom_frame = Frame(self.demo_frame,
    borderwidth=5,      relief=RIDGE,
    height=50,
    bg="cyan",
    ) ###
self.bottom_frame.pack(side=TOP, fill=X)


### Now we will put two more frames, left_frame and right_frame,
### inside top_frame.  We will use HORIZONTAL (left/right)
### orientation within top_frame.

# left_frame
self.left_frame = Frame(self.top_frame, background="red",
    borderwidth=5,      relief=RIDGE,
    width=50,
    ) ###
self.left_frame.pack(side=LEFT, expand=NO, fill=Y)


### right_frame
self.right_frame = Frame(self.top_frame, background="tan",
    borderwidth=5,      relief=RIDGE,
    width=250
    )
self.right_frame.pack(side=RIGHT, expand=YES, fill=BOTH)


# now put a button in each of the interesting frames
button_names = ["A", "B", "C"]
side_options = [LEFT, TOP, RIGHT, BOTTOM]
```

```
fill_options = [X, Y, BOTH, NONE]
expand_options = [YES, NO]
anchor_options = [NW, N, NE, E, SE, S, SW, W, CENTER]


self.buttonA = Button(self.bottom_frame, text="A")
self.buttonA.pack()
self.buttonB = Button(self.left_frame, text="B")
self.buttonB.pack()
self.buttonC = Button(self.right_frame, text="C")
self.buttonC.pack()
self.button_with_name = {"A":self.buttonA, "B":self.buttonB, "C":self.buttonC}

# now we some subframes to the buttons_frame
self.button_names_frame   = Frame(self.buttons_frame, borderwidth=5)
self.side_options_frame   = Frame(self.buttons_frame, borderwidth=5)
self.fill_options_frame   = Frame(self.buttons_frame, borderwidth=5)
self.expand_options_frame = Frame(self.buttons_frame, borderwidth=5)
self.anchor_options_frame = Frame(self.buttons_frame, borderwidth=5)

self.button_names_frame.pack(  side=LEFT, expand=YES, fill=Y, anchor=N)
self.side_options_frame.pack(  side=LEFT, expand=YES, anchor=N)
self.fill_options_frame.pack(  side=LEFT, expand=YES, anchor=N)
self.expand_options_frame.pack(side=LEFT, expand=YES, anchor=N)
self.anchor_options_frame.pack(side=LEFT, expand=YES, anchor=N)

Label(self.button_names_frame, text="\nButton").pack()
Label(self.side_options_frame, text="Side\nOption").pack()
Label(self.fill_options_frame, text="Fill\nOption").pack()
Label(self.expand_options_frame, text="Expand\nOption").pack()
Label(self.anchor_options_frame, text="Anchor\nOption").pack()

for option in button_names:
    button = Radiobutton(self.button_names_frame, text=str(option), indicatoron=1,
        value=option, command=self.button_refresh, variable=self.button_name)
    button["width"] = button_width
    button.pack(side=TOP)

for option in side_options:
    button = Radiobutton(self.side_options_frame, text=str(option), indicatoron=0,
        value=option, command=self.demo_update, variable=self.side_option)
    button["width"] = button_width
    button.pack(side=TOP)

for option in fill_options:
    button = Radiobutton(self.fill_options_frame, text=str(option), indicatoron=0,
        value=option, command=self.demo_update, variable=self.fill_option)
    button["width"] = button_width
    button.pack(side=TOP)

for option in expand_options:
    button = Radiobutton(self.expand_options_frame, text=str(option), indicatoron=0,
        value=option, command=self.demo_update, variable=self.expand_option)
    button["width"] = button_width
    button.pack(side=TOP)

for option in anchor_options:
    button = Radiobutton(self.anchor_options_frame, text=str(option), indicatoron=0,
        value=option, command=self.demo_update, variable=self.anchor_option)
    button["width"] = button_width
    button.pack(side=TOP)

self.cancelButtonFrame = Frame(self.button_names_frame)
self.cancelButtonFrame.pack(side=BOTTOM, expand=YES, anchor=SW)
```

```
        self.cancelButton = Button(self.cancelButtonFrame,
            text="Cancel", background="red",
            width=button_width,
            padx=button_padx,
            pady=button_pady
            )
        self.cancelButton.pack(side=BOTTOM, anchor=S)



        self.cancelButton.bind("<Button-1>", self.cancelButtonClick)
        self.cancelButton.bind("<Return>", self.cancelButtonClick)

        # set up the buttons in their initial position
        self.demo_update()


    def button_refresh(self):
        button = self.button_with_name[self.button_name.get()]
        properties = button.pack_info()
        self.fill_option.set  (  properties["fill"] )
        self.side_option.set  (  properties["side"] )
        self.expand_option.set(  properties["expand"] )
        self.anchor_option.set(  properties["anchor"] )


    def demo_update(self):
        button = self.button_with_name[self.button_name.get()]
        button.pack(fill=self.fill_option.get()
            , side=self.side_option.get()
            , expand=self.expand_option.get()
            , anchor=self.anchor_option.get()
            )


    def cancelButtonClick(self, event):
        self.myParent.destroy()


root = Tk()
myapp = MyApp(root)
root.mainloop()
```